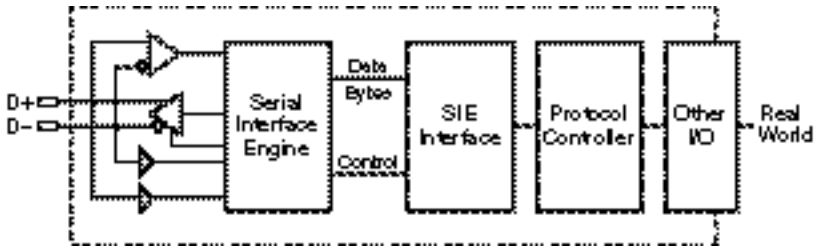
CHAPTER 4
CHOOSING A USB I/O DEVICE

In this chapter we transform the theoretical I/O device from Chapter 3 into several practical implementations. I used the word "choosing" in the chapter title to imply that we will be buying ready-made USB building blocks rather than starting one from silicon upwards. If you are a silicon designer then you should contact the USB Implementers Forum and inquire about their "Transceiver Macrocell Program." The I/O device designs described in this book will use some of the many commercially available USB components.

The block diagram of an I/O device, repeated in Figure 4-1 for your convenience, consists of several independent elements that I shall describe from the USB cable connection to the real world. These elements can be integrated into four basic catagories of device, and there are multiple variants within these catagories. The good news is that the ideal combination that you require will probably exist – the bad news is that you have to choose one!



<< Identify Transceiver in this diagram>>

Figure 4-1. Block diagram of I/O device

START WITH A TRANSCEIVER

A USB transceiver is required to meet the electrical characteristics of the bus. There are two fundamentally different types of transceiver component; the low/full speed transceiver is shown in Figure 4-2a and the full/high speed transceiver is shown in Figure 4-2b. These transceivers convert between the differential, bi-directional signaling on the bus to single-ended uni-directional, TTL/CMOS voltage levels for the Serial-Interface-Engine.

<< Include examples from NEC and Philips >>

Figure 4-2 Low/full speed and full/high speed transceivers

The low/full speed transceiver is a voltage/current level converter and produces a 1.5Mb/s or a 12Mb/s serial interface. Special bus conditions such as SUSPEND are detected and signaled to the TTL/CMOS interface. Note that the **directon** pin is driven by the TTL/CMOS interface – this transceiver needs to be told to listen or talk.

The full/high speed transceiver is also a voltage/current level converter and additionally includes the serial-to-parallel conversion and bit stuffing/unstuffing to produce a parallel interface. A 480Mb/s serial signal would be too fast for most logic families to deal with so this is converted to a parallel interface at 1 byte (60Mb/s), 2 bytes (30Mb/s) or 4 bytes (15Mb/s) wide depending upon the manufacturer.

Many specialized ASIC designs include all of the logic for a complete USB interface but use an external transceiver for its electrical parameters.

ADD A SERIAL INTERFACE ENGINE

An essential part of a USB interface is the Serial Interface Engine, or SIE. The SIE receives bits, or bytes, from the USB transceiver, validates them, and provides valid bytes to the SIE interface. The SIE must recover the data clock by resynchronizing its local clock with the transitions of the SYNC packet. This will ensure reliable reception of the PID and optional data elements of the packet. A high speed SIE also manages noise rejection with an active squelch circuit. Some initial SYNC transitions are lost as the squelch circuit turns off but the furthest receiver (five hub levels down) is guaranteed of receiving at least twelve SYNC transitions. Similarly, bytes are received from the SIE interface and transmitted serially onto the USB bus.

The SIE contains intelligence to manage the bus bit-level protocol including the bit-stuffing algorithm. A vendor can include more intelligence in the SIE if desired or can pass raw data with possible error status to the protocol controller. The implementation of the SIE interface varies with different vendor: some supply the bytes from USB in FIFOs, some in memory; some supply error status flags and expect the protocol controller to deal with error conditions, while some error-check the incoming data, implement the USB handshake protocol, and only interrupt the protocol controller once clean, validated data has been received. We'll look at a variety of USB components in the examples and will get first-hand experience of these implementation differences.

I am going to call a (USB transceiver + SIE) a USB peripheral. These components are used to add a USB connection to an existing design as shown in Figure 4-3.

<< Block diagram showing partitioning, ie uC with USB peripheral on a bus>>

Figure 4-3 Adding a USB peripheral

You can think of this USB peripheral the same way as you would treat a UART peripheral or a parallel port peripheral. It has control and status registers that the existing microcontroller, microprocessor or DSP must access and manipulate to generate and receive USB transmissions. This existing microcontroller, microprocessor or DSP already implements the "real world I/O" of Figure 4-1 and it will take on the additional role of USB protocol controller (examples are included in later chapters).

A USB peripheral has three main attributes that will guide your choice of a device, as shown in Figure 4-4.

Figure 4-4 Choosing a USB peripheral

A major decision will be the required data throughput that is required for your I/O device. Although not always true, a faster connection typically costs more! The low-speed option of USB was added specifically to support low cost devices such as mice or low rate data acquisition devices. The interface to USB can be one of three standard speeds – 1.5Mb/s, 12Mb/s or 480Mb/s. There are two popular interfaces to the microcontroller system – a serial I2C connection and a parallel bus connection. A I2C connection is low cost but also low throughput – but if it can support your required data rate this is an easy choice. There is a range of parallel bus implementations with master-mode DMA included on some vendor's components for high throughput. An 8 bit parallel bus is sufficient for a full-speed peripheral but this translates to a 60MHz signaling rate for a high speed USB connection. A 16-bit, or 32-bit, parallel bus is a better choice here.

The third, and largest, variable to consider when choosing a USB peripheral is the number and capabilities of USB endpoints supported. A simple device can be implemented with a Control endpoint and a single interrupt endpoint. A bulk or isochronous endpoint needs high speed static RAM to support data transfers and this is an expensive commodity on a silicon chip. Some peripheral components

therefore only offer 3 or 4 additional endpoints for lower cost while some manufacturers offer 8 to 12 for higher functionality.

The choice can be a little over whelming at this time – lets wait until we have implemented several examples to that we can get a feel of what's needed.

An important benefit of using a USB peripheral is that your choice of microcontroller or microprocessor is independent of the USB peripheral component. If you have libraries of existing software for a particular microcontroller then you can continue to use it, and add some USB communications libraries! The next section will discuss integrated devices and the microcontroller choice is more limited.

Several USB peripheral components are available and a selection are shown in Figures 4-5 and 4-6.

<< Use Philips D11 and D12, Netchip 2890 and Nationals 9603>>

Figure 4-5. Low/Full-Speed USB peripherals

<< Include NEC, Philips, Netchip>>

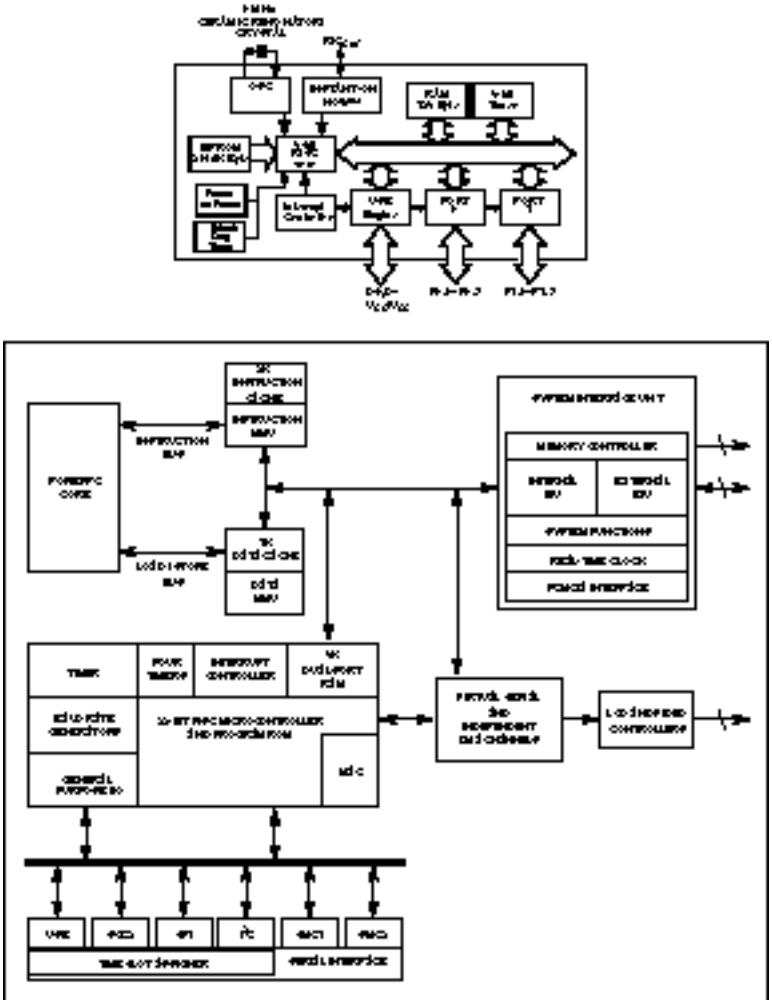
Figure 4-6. High/Full-Speed USB peripherals

Add a Protocol Controller

The protocol controller was an external microcontroller, microprocessor or DSP in the previous section. A very popular implementation for USB is to integrate a programmable microcontroller, a USB peripheral and some dedicated I/O devices, into a single "USB microcontroller". Many microcontroller suppliers have intgrated the USB peripheral into their family of devices to create a single-chip USB controller. Different vendors use different microcontrollers so different instruction sets are used and you should check if your favorite microcontroller comes with a USB "option".

There is a large span of program memory space varying from 0.5K to 32K. Most vendors produce a family of devices so you can better match your requirements. The microcontroller's program memory may be ROM, EPROM, EEPROM or even RAM. If ROM is used, then an external, serial EEPROM is typically used to customize a reasonably fixed I/O device configuration. EPROM devices are programmable, and some versions are shipped in a plastic package for lower cost—these sealed devices are not erasable and are referred to as One-Time-Programmable (OTP). An EEPROM, sometimes called a flash device, is programmable in-circuit. Some USB microcontrollers use RAM for program memory; this must be downloaded via USB or from an attached EEPROM before the microcontroller can be brought out of reset. These differences will become clear once we discuss the development environments in Chapter 5. Each microcontroller will also have data memory space and may have a selection of internal devices such as interrupt controller, timers, counters, etc.

For microcontrollers, there is an even wider diversity of implementations available, ranging from a simple 8-bit implementation such as the Microchip PIC device to the 32-bit Intel StrongArm device (Figure 4-7).



<< Replace PowerPC with StrongArm SA1110>>

Figure 4-7. Wide range of USB microcontrollers available

Finally, Add Real World I/O

The density and complexity of the integrated I/O capability also vary greatly. It is best to design your I/O device first, and then look for the best match of features from a list of available components.

A representative low-speed device is shown in Figure 4-8. The PICC745/765 from Microchip Technology Inc. is towards the high end of this category with ample real-world I/O features such as 5 or 8 channels of 8-bit A/D, 22 or 33 I/O pins, 3 timers, 2 PWM modules and a USART to complement the 6 endpoints that a low-speed device is allowed to have. A low-speed device supports only control and interrupt transfers, but this is enough to implement all of the HID examples in Chapter 7 and most of the enhancements in Chapter 9.

<< Copy from PCI datasheet >>

Figure 4-8. Representative low-speed device

I personally found that PIC instruction set quite bizarre and after many attempts could not get the simplest of examples working. Fortunately a reader from book one came to my rescue and implemented the PIC examples in Chapter 7. The breadth of the PIC family is enormous (take a look at www.microchip.com) and there is a wealth of software available for this family. The addition of a USB port will further expand this application range.

A representative full-speed device is shown in Figure 4-9, it is the 8051-based EZ-USB from Cypress Semiconductor. This component has a unique implementation in that the program memory of the protocol controller is RAM. On power-up, an intelligent SIE holds the protocol controller in reset. The SIE understands the enumeration process and can complete the process without help

from the protocol controller. The device driver specified by the EZ-USB device descriptor knows how to do one thing: download a program into the program RAM and remove the reset from the protocol controller. We thus soft-load a program into the I/O device!

The EZ-USB component then programmatically detaches itself from the hub and reattaches itself with its newly loaded personality. Cypress calls this process "renumeration," and it means that the device doesn't have to be built with a mask ROM or even be "programmed". If you think that your I/O device program may change after it ships to users, then software update is as easy as providing a new file on the PC host. No product to recall. No new parts or EPROMs to manufacture and supply. Just ask your users to download an update from the Internet!

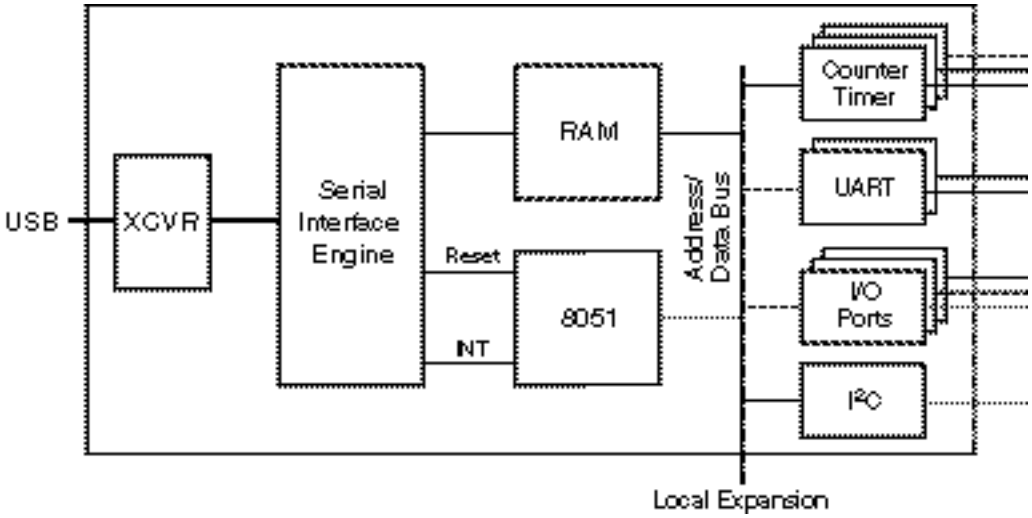


Figure 4-9. Representative full-speed device

The EZ-USB has a full implementation of USB endpoints including double-buffering of bulk endpoints. This ample supply of features enables it to easily support our design examples.

The EZ-USB component is a little more expensive than ROM or EPROM parts because its program in RAM means that the die is larger. For low- and medium-volume projects, however, use of this component could be a cheaper solution overall. Cypress has compatible ROM parts for high volume—these are not downloadable, but you are not penalized for designing a successful product!

The EZ-USB uses the 8051 microcontroller to move bytes of data between its endpoint FIFOs and the real world. It includes some additional features over-and-above the standard 8051 microcontroller to speed these transfers but this strategy is only suitable for full-speed designs. In preparation for high-speed data transfers Cypress added a microprogrammable DMA engine to the EZ-USB base architecture and called this the FX (the FX also runs twice as fast and has a variety of other enhancements, but the DMA engine is the key element for higher speed data movement). Figure 4-10 shows the EZ-FX component with the General Purpose InterFace (GPIF, nee micro-programmable DMA engine) and slave FIFOs added. The 8051 core is responsible for setting up the DMA data transfers but is not involved in the data movement.

<<Copy from FX TRM >>

Figure 4-10. The EZ-FX moves data with a DMA engine

The Cypress EZ-FX2 had the same architecture as the EZ-FX but the SIE runs at 480Mb/s and the data FIFOs are much larger and move programmable. An FX application can be simply moved to an FX2 and immediately benefit from the higher bus speed (we'll do this in Chapter 10).

Kawasaki took an innovative approach with their real world I/O as shown in Figure 4-11. They have a 16-bit embedded RISC processor and a standard collection of I/O devices including a 10-bit A/D converter and PWM outputs with external expansion capability. They also have a customer-configurable logic array (up to 8K gates) that can be programmed to need your I/O requirements. Kawasaki provides a development board (Figure 4-12) where your custom I/O application is debugged externally before committing your design to a masked part. In this way, you get the benefits of a proven USB core design in which you can customize the I/O to create a single chip solution.

<< Copy from USB 3 User Manual >>

Figure 4-11. Fully programmable I/O block

Kawasaki uses this core to create a range of fixed-function parts which we shall review later in this chapter. A 480Mb version of this configurable core has been announced and Kawasaki has migrated several of their network devices to this higher speed device.

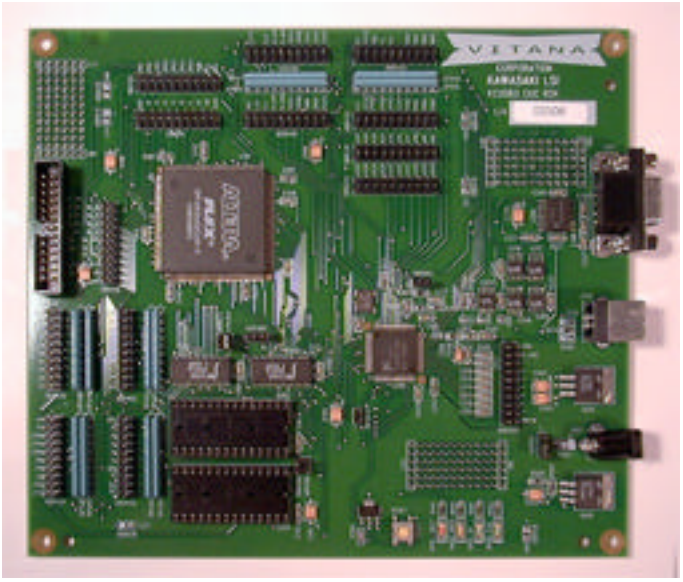


Figure 4-12. Development board for custom I/O

CUSTOM ASIC

My next category is to implement your design as a single chip (or two chips, ASIC + transceiver) by using a synthesizable core. Several vendors, such as www.in-system.com, www.lucent.com and www.trenz-electronic.de, offer VHDL or Verilog libraries which you can augment to include your custom I/O features. If you intend to sell many thousands of your I/O device then I recommend that you design and debug it first using a reprogrammable microcontroller and then contact one of these ASIC vendors who will assist you in porting your design to their process.

FIXED FUNCTION DEVICES

This chapter has presented reprogrammable designs up until now. Any one of these designs could be converted into a fixed-function device by also supplying integrated firmware. A vendor will look at the market size for a particular function and may well cost-optimize their general-purpose solution for this dedicated application. Some of the first fixed-function devices were USB-to-Serial and USB-to-Parallel converters – we will see in later chapters that these functions are straightforward to design using a reprogrammable USB microcontroller but they can be made cheaper by optimizing their function.

There is now a wide selection of fixed-function devices, some of which are shown in Figure 4-13. All these components are interesting and are discussed in more detail in "Building I/O Bridges", Chapter 10.

Figure 4-13. Fixed-function USB devices

SELECTION CRITERIA

As I said, the good news is that there is a wide selection of available building blocks with which to start your design. But how do you choose?

Decision zero is "is the device I want to design already available". It will almost always be cheaper to buy a solution or near-solution than design your own.

Your first decision should choose the bus speed that you need to operate at – it will generally be cheaper to use low-speed, more expensive for full speed and more expensive again for high speed. The big question is how large a data rate does your I/O device require? Figure 4-14 recaps the maximum data rates for each bus speed.

Bus Speed	Maximum Data Rate
Low	800 Bytes/second
Full	64K Bytes/second
High	Bytes/second

Figure 4-14. Choosing an implementation speed

It is technically possible to build a mouse to operate at full or high speed but what is the point? We are tracking the movement of a person's hand and 800B/s is way more than we need. At the same time, it is possible to interface a 40GB hard drive using a low-speed connection – you would fall asleep while waiting for files to transfer!

Your I/O device data rate will select a bus speed.

Your second decision will depend upon the capability of your I/O device. It probably has a microcontroller or a microprocessor embedded in it (if not, it soon will have, a USB I/O device almost always requires a microcontroller). If your embedded microcontroller, microprocesssor or DSP has a family member with a USB port then **this** is the first place that you should look for a USB-based solution. If a USB-aware family member is not readily available then you should choose a USB peripheral component; most manufacturers supply the underlying low-level USB communications code as an example (many are included in Chapter 9 on the CDROM) and, with the help of this book, you should be able to integrate this code into your project (there are many examples in the second half of this book).

Some of you may choose to add another microcontroller to your I/O device and treat it as a "USB subsystem". A private communication path between your embedded microcontroller, microprocessor or DSP and the USB microcontroller would be set up and all USB traffic would be handled by this additional microcontroller. This may be a more expensive solution initially but it is quicker to implement since less new software has to be written. The design could be cost-optimized once it is successful in an earlier marketplace. USB microcontrollers with 8-bit architecture, 16-bit and 32-bit are all available.

Your third decision is based on the I/O complement that you require for your I/O design (yes, this is heavily linked to decision 2). Of the available components that meet my needs, which one is the closest to my design. Can I do this with a single chip, or customizable chip?

Other decisions include ease of development (the subject of the next chapter), price and availability. These all start to blur after the first three decisions and your individual situation may be different. As they say in all advertisements "your mileage may vary".

CHAPTER SUMMARY

I hope that I have given you a flavor of what is available and some key decisions you should make to move forward. I have not addressed the effects of small, medium or large volumes since I have discovered that every company I have dealt with has a different view of these criteria. There are a large number of USB component vendors who want your business – this can only improve product capability and reduce costs. So what do you want to do?